Klassen und Objekte

- Motivation
- Bankkonten und ihre Implementierung
- Begriffe

Zustandslose Methoden

Fast alle bisherigen Methoden sind nur abhängig von ihren Parametern; Beispiele

- boolean[] calculatePrimes(int n)
- String reinigeNachricht(String nachricht)
- String kodiereNachricht(String gereinigteNachricht)
- String versteckeNachricht(String nachricht, String traegerMedium)
- String zeigeNachricht(String steganogramm)
- String toUpper(String text)
- int scan(String text, String needle)
- String[] split(String text, char delimiter)

Sie nicht zum Beispiel *nicht* abhängig von vorherigen Aufrufen

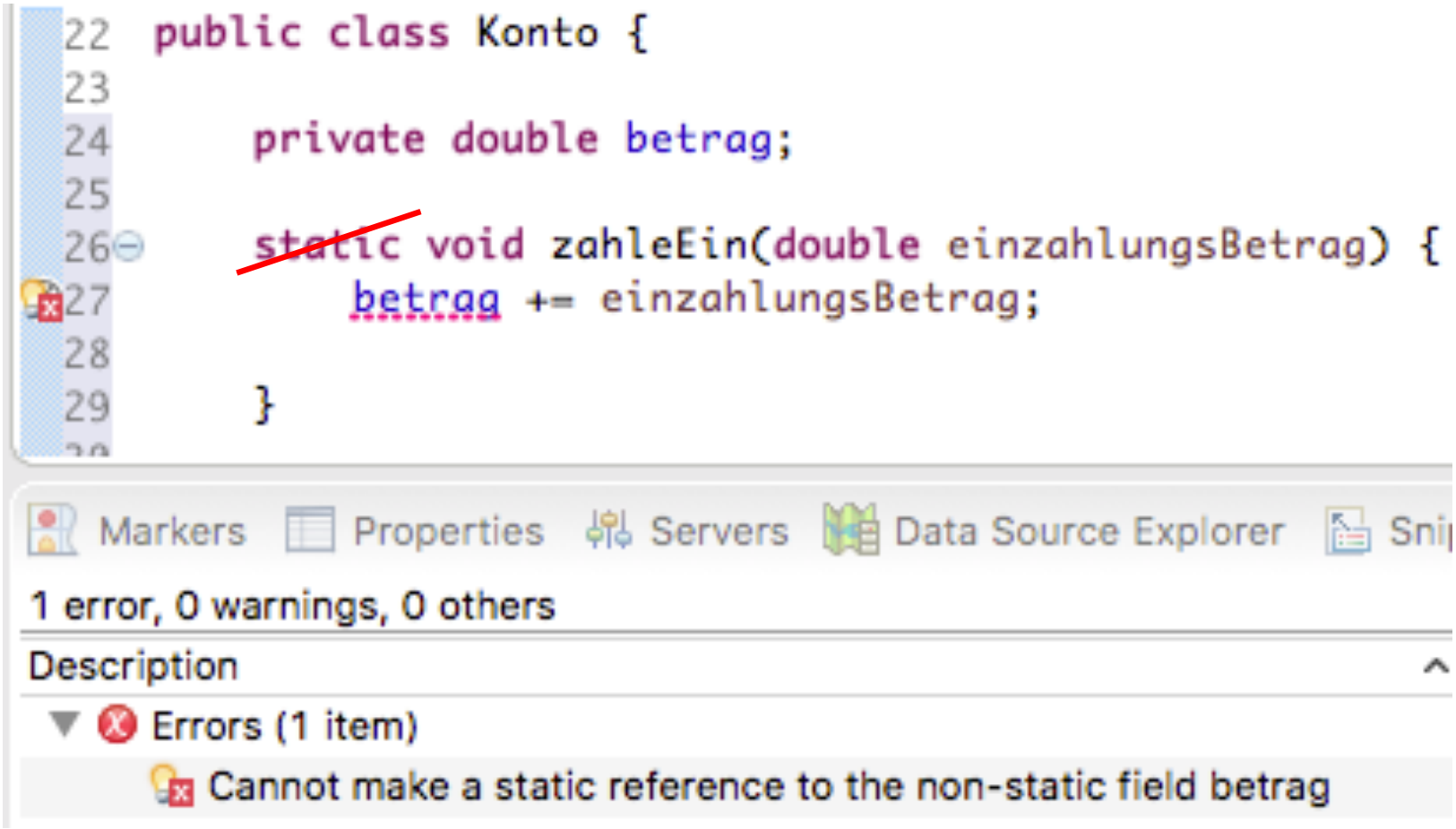
Jetzt aber ein anderes Beispiel: Wie ist das mit einem Bankkonto?

- Welche Eigenschaften hat ein Bankkonto?

- Was kann ein Bankkonto, welche Operationen (Methoden) lassen sich darauf anwenden?

Warum sind die Methoden nicht „static“?

Illustration



Auf (nicht-statische) Felder einer Klasse kann nur von nicht-statischen Methoden aus zugegriffen werden!

Warum nicht auch die Felder „static“ vereinbaren?

```
private static double betrag;

static void zahleEin(double einzahlungsBetrag) {
    betrag += einzahlungsBetrag;

}

static double kontostand() {
    return betrag;
}

public static void main(String[] args) {

    KontoStaticDemo k1 = new KontoStaticDemo();
    k1.zahleEin(100);
    println(k1.betrag);

    KontoStaticDemo k2 = new KontoStaticDemo();
    k2.zahleEin(200);
    println(k2.betrag);

    println(k1.betrag);

}
```

Achtung: Warnungen!

```
<terminated> KontoStaticD
100.0
300.0
300.0
```

Warum nicht auch die Felder „static“ vereinbaren?

```
private static double betrag;

static void zahleEin(double einzahlungsBetrag) {
    betrag += einzahlungsBetrag;

}

static double kontostand() {
    return betrag;
}

public static void main(String[] args) {

    KontoStaticDemo k1 = new KontoStaticDemo();
    k1.zahleEin(100);
    println(k1.betrag);
```

Achtung: Warnungen!

The static field KontoStaticDemo.betrag should be accessed in a static way

The static method zahleEin(double) from the type KontoStaticDemo should be accessed in a static way

```
    println(k1.betrag);

}
```

Warum nicht auch die Felder „static“ vereinbaren?

```
private static double betrag;

static void zahleEin(double einzahlungsBetrag) {
    betrag += einzahlungsBetrag;

}

static double kontostand() {
    return betrag;
}

public static void main(String[] ar

    KontoStaticDemo k1 = new KontoSt
    k1.zahleEin(100);
    println(k1.betrag);
```

Folgende Möglichkeiten bestehen für einen „statischen Zugriff“:

```
KontoStaticDemo.zahleEin(100);
println(KontoStaticDemo.betrag);

zahleEin(100);
println(betrag);
```

Man sieht, dass gar kein Bezug auf ein spezielles Konto genommen wird

-> Es existiert nur *ein Feld pro Klasse*, d.h. *nicht* ein Feld pro Objekt!

The static field KontoStaticDemo.betrag should be accessed in a static way

The static method zahleEin(double) from the type KontoStaticDemo should be accessed in a static way

```
    println(k1.betrag);

}
```

Erklärung: das Schlüsselwort „static“ 1/2

- Wir kennen Konstanten, z.B.

```
static final double PI = 3.141592653589793;
```

- Konstanten gibt es bei der Programmausführung genau *ein Mal* in einem Programm (für Konstanten ist das sinnvoll)
- Dafür sorgt das Schlüsselwort „static“
- Daher geht das bei den Bankkonten schief:
  Es gibt bei der Programmausführung genau einen Inhaber, einen Kontostand und eine Kontonummer, die sich alle Konto-Objekte, die erzeugen, teilen!
- Ohne „static“ bekommt jedes Objekt seinen eigenen Inhaber, seinen eigenen Kontostand und seine eigene Kontonummer

Erklärung: das Schlüsselwort „static“ 2/2

- Daher geht das bei den Bankkonten schief:
  Es gibt bei der Programmausführung genau einen Inhaber, einen Kontostand und eine Kontonummer, die sich alle Konto-Objekte, die erzeugen, teilen!

```
private static String inhaber;
private static float betrag;
private static int nummer;
```

- Ohne „static“ bekommt jedes Objekt seinen *eigenen* Inhaber, seinen *eigenen* Kontostand und seine *eigene* Kontonummer

```
private String inhaber;
private float betrag;
private int nummer;
```

Konsequenz

- Mit „normalen“ *Feldern* (also nicht-*static*) kann jedes Objekt seinen eigenen individuellen Zustand zwischen zwei Methodenaufrufen erhalten
- Dadurch wird es möglich, zustandsabhängige Methoden zu erstellen:
  Vorherige Aufrufe hinterlassen einen (End-)Zustand,
  der für spätere Aufrufe als Ausgangszustand dient

- Beispiel
  Geld auf ein (spezielles) Bankkonto einzahlen, das später wieder abgehoben werden kann

Beispiel (Fortsetzung): Wie ist das mit einer Bank?

- Welche Eigenschaften hat eine Bank?

- Was *kann* eine Bank, welche Operationen (Methoden) lassen sich darauf anwenden?

Viele Konten in der Bank

- Programmiert haben wir Klassen

```
class Konto { ... }
class Bank { ... }
```

- Eine Bank hat *viele* Konten;
  diese stellen wir dar als *Objekte* nach dem *Muster ihrer Klasse*
- Wie bei Arrays erzeugen wir zur Laufzeit des Programms ein *neues* Konto mit „new“ auf dem Heap:

```
new Konto()
```

Erzeugen von Konten

- Wie bei Arrays erzeugen wir zur Laufzeit des Programms ein *neues* Konto mit „new“ auf dem Heap:

```
new Konto()
```

- Wie bekommen wir z.B. den Inhaber und die Kontonummer (falls die *im* Konto gewünscht ist) in das neue Konto-Objekt hinein?
  → als Parameter

```
new Konto( “Meier, Kurt“, 1234567890 )
```

Erzeugen von Konten

- Wie bekommen wir z.B. den Inhaber und die Kontonummer in das neue Konto-Objekt hinein?
  → als Parameter

```
new Konto( “Meier, Kurt“, 1234567890 )
```

- Jetzt müssen wir dafür sorgen, dass die Information im Konto gespeichert wird
  → Wir benötigen einen *Konstruktor* für die Klasse „Konto“

```
Konto( String kontoInhaber, int kontoNummer ) {
        inhaber = kontoInhaber;
        nummer = kontoNummer;
}
```

- Genau dieser Konstruktor wird mittels „new Konto(...)“ oben aufgerufen und liefert das neue Objekt als Ergebnis

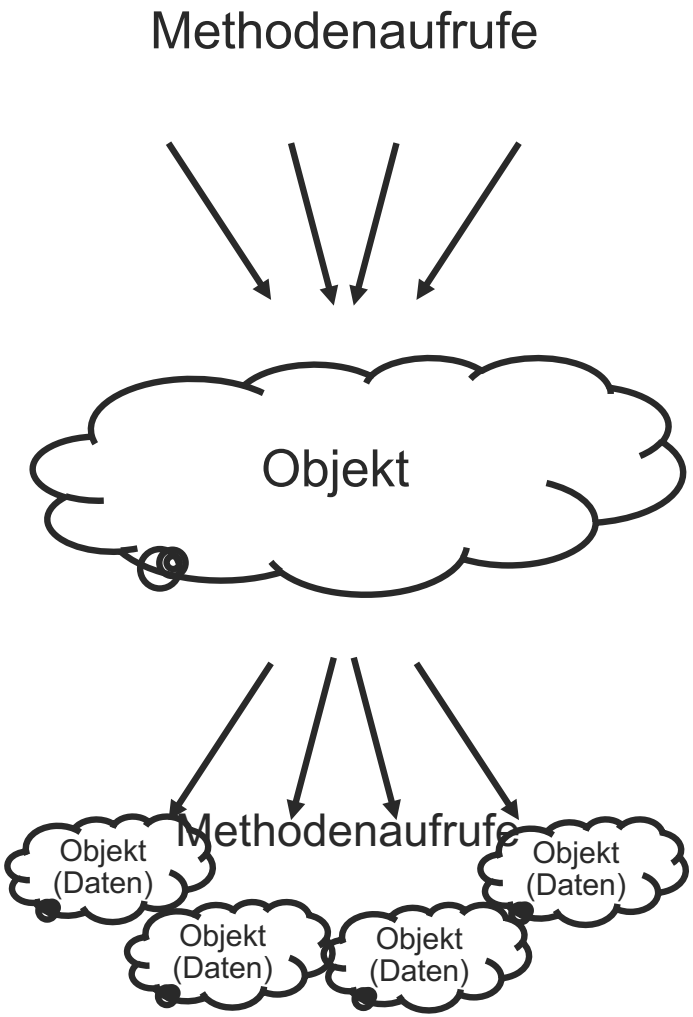
Ein mögliches Hauptprogramm

```
public class BankBeispiel {

    public static void main(String[] args) {
        Bank bank = new Bank();

        int müllerNummer = bank.eröffneKonto("Müller");
        int meierNummer = bank.eröffneKonto("Meier");

        bank.zahleEin(müllerNummer, 200f);

        bank.zahleEin(meierNummer, 100f);

        bank.listeKontenAuf();

        bank.zahleAus(müllerNummer, 200);
        bank.schließeKonto(müllerNummer);

        bank.listeKontenAuf();
    }

}
```

```
<terminated> BankBeispiel [Java Appli
0: 200.0 (Müller)
1: 100.0 (Meier)
0: 0.0 (Müller) IST GESPERRT
1: 100.0 (Meier)
```

Konzept der objektorientierten Programmierung

- *Objekte* stehen *im Mittelpunkt*
- *Nur* mittels *Methodenaufrufen* können wir mit ihnen arbeiten, z.B. etwas abfragen oder eine Zustandsänderung bewirken



Objekte haben Zustände: Beispiel

Ein Aufzug als Objekt:

Was kann man mit einem Aufzug machen?
Welche Operationen hat er/braucht er?

Welche Zustände kann er haben?

Perspektive des *Benutzers*

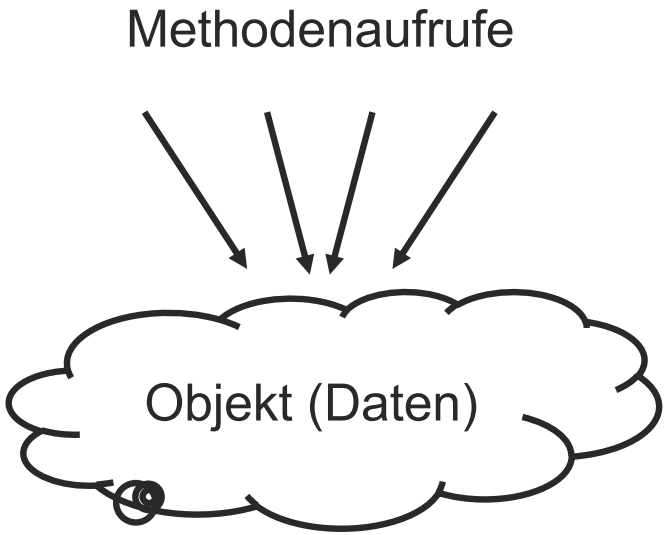
Wir sagen:
Aufzug, fahre in den 8. Stock!

Wir ziehen den Aufzug ***nicht*** selbst!
Nur der Aufzug selbst kann seinen Zustand ändern!

Zustandsänderung, Umgang mit Objekten

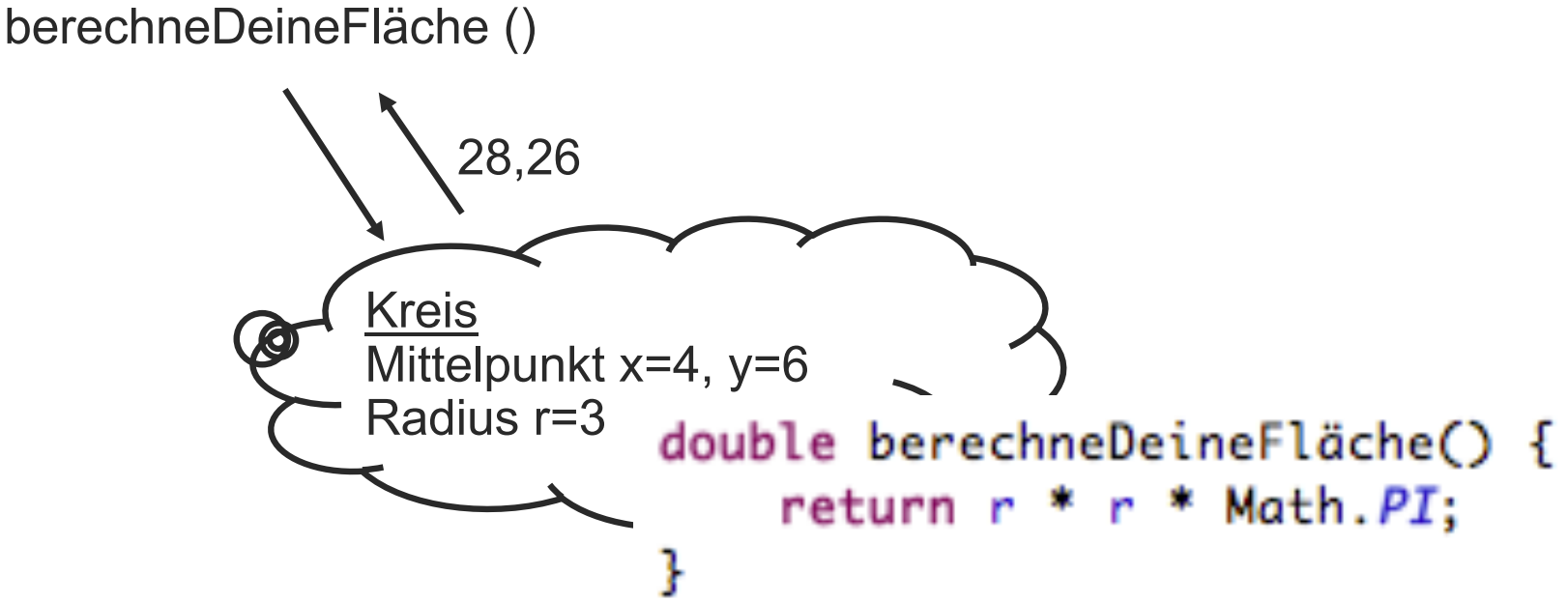
Objekte stehen *im Mittelpunkt*,
Methodenaufrufe bewirken eine ***Zustandsänderung***

Lokalitätsprinzip:
Alles was mit den Daten arbeitet (und nur das), wird zusammen in einer Klasse implementiert, z.B. Klasse „Aufzug“ mit Zustandsinfo und Methoden



In der Programmiersprache: Beispiel für Objekte 1/2

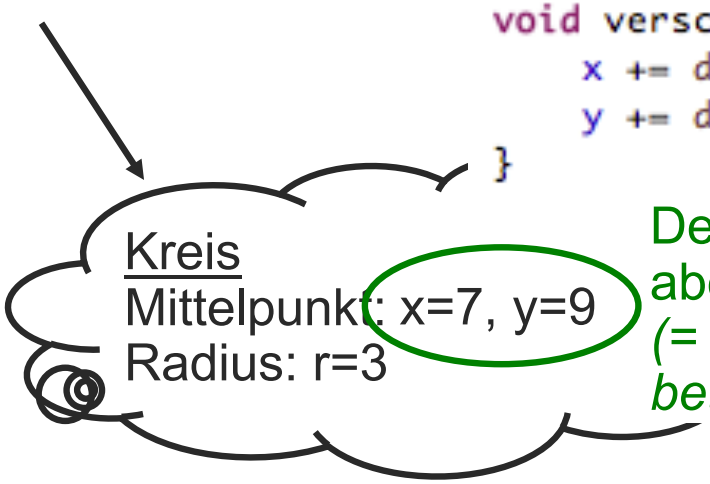
Daten (Objekte) stehen im Mittelpunkt, man kann ihren Zustand abfragen / sie mit ihren Zustandsdaten arbeiten lassen



In der Programmiersprache: Beispiel für Objekte 2/2

Daten (Objekte) stehen im Mittelpunkt,
Methodenaufrufe bewirken eine Zustandsänderung

verschiebeDich (3, 3)




```
void verschiebeDich(double dx, double dy) {
    x += dx;
    y += dy;
}
```

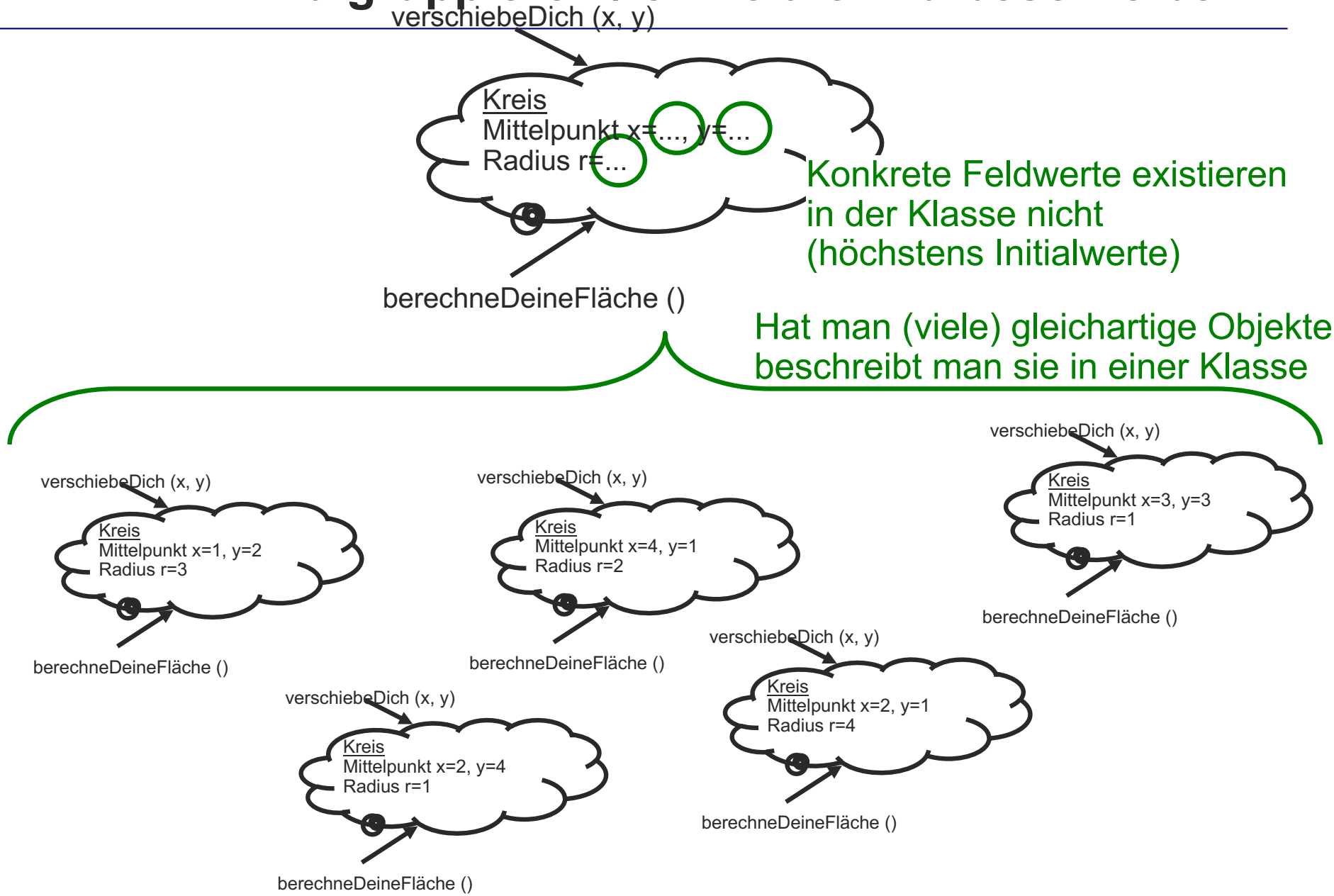
Der Objektzustand wird geändert, aber ***nur vom Objekt selbst*** *(= von Methoden der Klasse des betroffenen Objekts)*!

Wir wenden uns an den Kreis, in Java-Schreibweise:

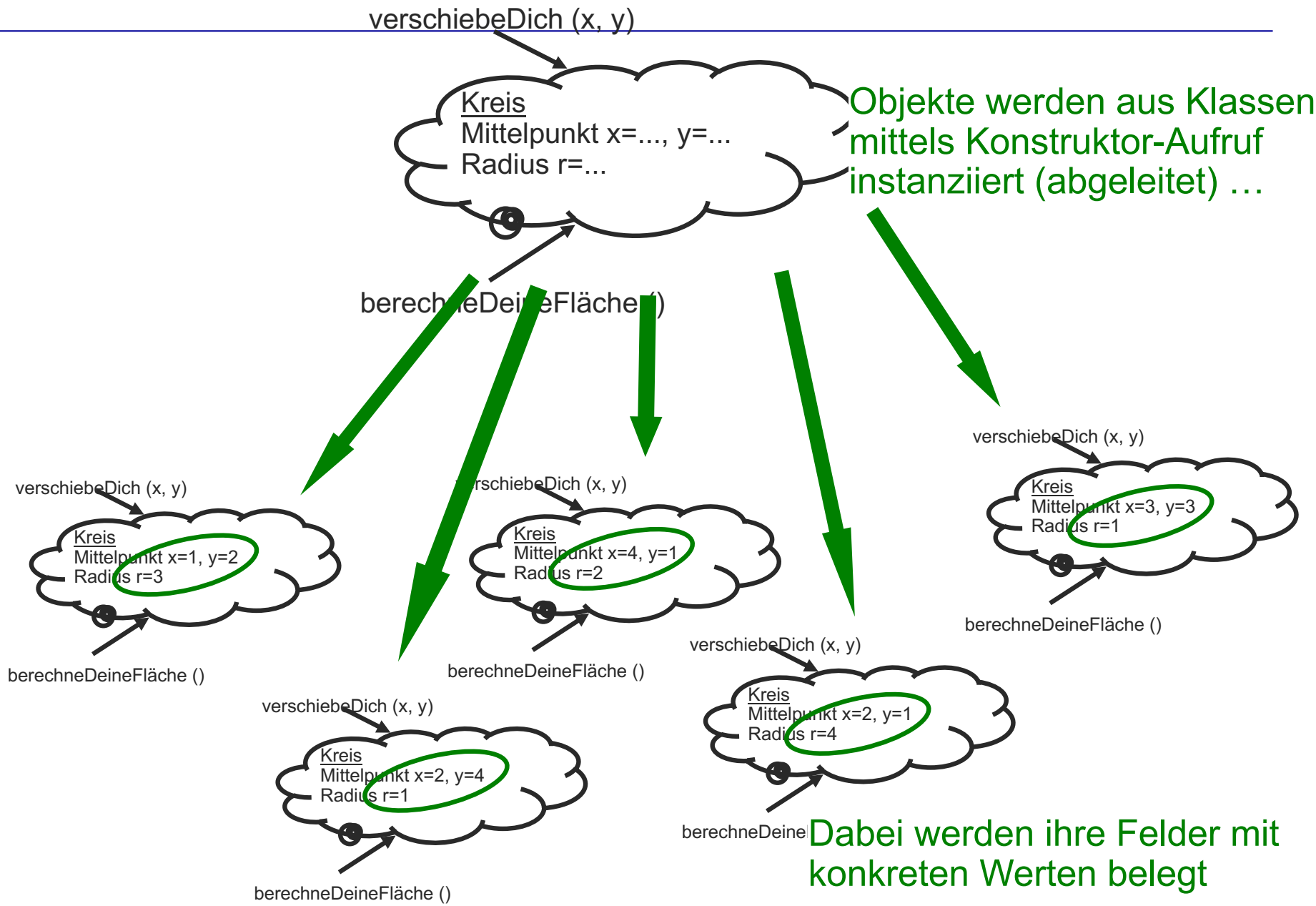
```
Kreis k = new Kreis();
k.verschiebeDich( 3, 3 );
```

Daten und die zugehörigen *Methoden* werden in einer Klasse *zusammengefasst*, *nur* die Methoden einer Klasse selbst greifen auf ihre Felder zu (können zugreifen)

Man benutzt Klassen, um gleichartige Objekte zu gruppieren / einheitlich zu beschreiben



Zur Laufzeit eines Programms existieren nicht die Klassen (d.h. die Java-Dateien), sondern nur deren Objekte (Instanzen)



Klassen und Objekte

- Eine ***Klasse*** *beschreibt* eine (möglicherweise sehr große) Menge *gleichartiger Objekte* mit individuellenZuständen (= individuelle Feldwerte: jeder Kreis liegt an einem anderen Platz, hat eine andere Größe)

- Jedes ***Objekt*** ist eine eindeutige Instanz seiner Klasse
  Es *kennt seine Klasse*